KIS Administrator's Guide

Kabira Infrastructure Server

KIS Administrator's Guide : Kabira Infrastructure Server

Published 27 Sep 2010

This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you! Feedback about this document should include the reference KISAG-32.2 . Updated on 27 Sep 2010 .

Contents

List of Figures

1

Introduction

This chapter provides an overview of how you manage Kabira Infrastructure Server (KIS) applications and the tools that you use to manage and monitor them. Later chapters of the book expand on these basic ideas.

This chapter contains the following sections:

- the section called “ Deploying applications ” on page 1
- the section called “ Monitoring shared memory ” on page 2
- the section called “ User Environment ” on page 3

Deploying applications

KIS applications are comprised of one or more KIS components running as a set of processes, called engines. The engines run on nodes that provide all the runtime services and management infrastructure necessary to execute the components.

To deploy a KIS application, you need to:

1. set up a node (or nodes for a distributed KIS application)
2. create engines on a node
3. configure your engines
4. start your engines

Setting up a node

KIS provides one simple command-line tool, `swsrv`, that allows you to create, start, stop, and remove nodes. Through command-line options, you can configure node name, memory size, secure access, listening ports and hosts, and installation directory.

For more details on nodes, see Chapter 2.

Creating engines

A KIS engine consists of one or more application components, like those you build with the KIS Design Center. You deploy a component by assigning it to an engine. An engine is a process that provides an execution context for the component. A single component can be assigned to one or more engines.

You define the assignment of component to engine in a deploy specification. Deploy specifications define:

- engines
- the components they comprise
- application-specific node and engine properties

The actual assignment takes place when you load a deploy specification with one of the engine management tools, the graphical Engine Control Center or the command-line tool `swnode`.

For more on deploy specifications and the engine management tools, see Chapter 3.

Configuring engines

Typically, you define engine configuration in the deploy specification by specifying engine and node properties. When you load the deploy specification, the engine management tools, configure the engine as specified.

However, the engine management tools also allow you to configure engine and node properties after engines are installed.

On configuring engines with the engine management tools, see Chapter 3.

Controlling engines

The engine management tools allow you to start and stop individual engines, a group or engines, or all engines on the node. For more details, see Chapter 3.

Monitoring shared memory

The KIS Monitor lets you examine details of system operation. It interactively decodes shared memory on the local node to let you examine KIS types and instances. For details on the KIS Monitor, see Chapter 4.

User Environment

To use the tools described in book, the user must have the following two environment variables set, typically in their shell profile:

SW_HOME	This must be the directory where KIS was installed.
PATH	The user's path should include `$SW_HOME/distrib/` platform `/scripts` and `$SW_HOME/distrib/` platform `/bin` , where platform is `sol` for Solaris or `hpux` for HP-UX.

2

Managing Nodes

This chapter describes how you install, configure, and manage a Kabira Infrastructure Server (KIS) node.

It comprises the following sections:

- the section called “ What is a KIS node? ” on page 5
- the section called “ Node command-line tool ” on page 6
- the section called “ Node configuration ” on page 10

What is a KIS node?

A KIS node consists of a shared memory region (with its associated shared memory file) managed by a System Coordinator. The System Coordinator manages the engines, providing configuration and control services.

System initialization

When the System Coordinator starts, it reads the configuration information stored in the registry, attaches to shared memory, and starts its engines as described in the following paragraphs.

Reading Configuration Information The System Coordinator reads its configuration from the registry; the registry key is the node name specified on the command line. By default, this is the machine name returned by `uname -n` (see the section called “ Node command-line tool ” on page 6).

Each node has its own runtime directory, to avoid name conflicts with log files, etc. By default, the node name specifies the installation directory; you can change this with `swsrv -p` command-line option (see the section called “ Node command-line tool ” on page 6).

Attaching to shared memory The System Coordinator attaches to the shared memory file specified in the registry. If the specified file does not exist, the System Coordinator creates it, allocates memory, and initializes it using the configuration information from the registry. If the shared memory file does exist, the System Coordinator performs recovery on it to make sure that it is in a valid state. At this time, the System Coordinator kills all processes attached to shared memory and to the registry as part of its startup (you can control this with the `swsrv` -k flag).

Starting engines The System Coordinator starts all engines that are configured to auto-start. You can control this with the engine property `autoStart` (see "Generic engine properties" on page 50). The System Coordinator gets the engines' executable name and command-line arguments from the registry.

Recovery

When a KIS engine fails, the System Coordinator

1. abruptly stops all engines (since the system is in an unstable state)
2. releases all resources
3. performs recovery by rolling back any active transactions
4. restarts all engines that were running prior to recovery whose `restart` engine property is `true` .

No operator intervention is required for system recovery to take place.

For external engines, you can configure whether recovery initiates when the engine fails (`triggersRecovery`) and whether the engine is part of recovery when another engine fails (`needsRecovery`).

Node command-line tool

The node command-line tool `swsrv` allows you to create, start, stop, and remove nodes.

When you start an existing node with `swsrv -c start` , it

1. updates the node with any user provided command line configuration options
2. starts the System Coordinator
3. installs system engines
4. starts system engines
5. lets the System Coordinator initialize the node

If a node does not already exist, the `start` command creates one first and then performs the steps above.

swsrv also redirects standard output from the System Coordinator to coordinator.stdout, which is where you should look for log output.

swsrv usage

```
swsrv -c command [-d] [-H path] [-N node name]
                  [-h name server host] [-k] [-l web port]
                  [-m memory size] [-n name server port]
                  [-P port] [-p path] [-S web host]
                  [-s host] [-v] [-w password] [-W password]
  -c      configure clobber start stop forcestop status
```

The following table describes each of the `swsrv` commands.

Command	Description
configure	Configure a KIS node with the supplied parameters, but stop the node after configuration is complete. You may use configure to install a new node, or to reconfigure an existing node. The node must be stopped in order to be reconfigured.
clobber	Do a forcestop, and then remove the node runtime directory.
start	Start a node. If the node does not already exist, it is first configured. If configuration parameters are supplied to an existing node, the node is first reconfigured.
stop	Stop a node. The node is shutdown cleanly.
forcestop	Do not attempt a clean shutdown. Terminate the coordinator forcefully if required.
status	Return the status of the node.

The following table describes the general options that are valid with all commands.

Option	Description
-d	Enable debugging. Normally, enable this only if requested by Kabira support.
-H	Specify location of SW_HOME. Normally the $SW_HOME environment variable is expected to contain this information.
-N	Specify an alternate node name. Normally the node name is the machine name returned by `uname -n` . Note: If an alternate node name is specified, it will be required for all subsequent commands to swsrv for this node!
-p	Specify an alternate run time path. Normally a subdirectory is created using node name. Note: If an alternate run time path is specified, it will be required for all subsequent commands to swsrv for this node!
-w	Specify node password. If a node password has been configured (see -W), the -w option is required.

The following table describes the configuration options valid for the `configure` and `start` commands.

Option	Description
-h	Specify java name server host. The user can specify an alternate name server running on another host. Normally, a Java name server running under the coordinator is used.
-k	Disable termination of processes attached to shared memory and the registry during recovery. Normally, the coordinator kills all processes attached to shared memory and the registry when it starts and performs initial recovery. This is to ensure there is nothing accessing the system when it is recovered. This is required for correct operation of the KIS node, so this option should not be used unless directed by Kabira support.
-l	Set web server port. The user can specify a specific port, which must be unused. A web server will be started on this port.
-m	Set shared memory size in Mb. Note that on Solaris, making shared memory size a multiple of the largest available system page size will result in the best system performance. The supported page sizes can be found using the Solaris 'pagesize -a' command.
-n	Set java name service port. Normally a free port is randomly chosen. The user can specify a specific port, which must be unused. A java name server will be started on this port.
-P	Set coordinator port. Normally a free port is randomly chosen. The user can specify a specific port, which must be unused. The coordinator will listen on this port.
-S	Set web server host name. If a host has multiple network interfaces, this may be used to bind the web server port to a specific interface. The host name must be a valid host name on the local machine.
-s	Set coordinator host name. If a host has multiple network interfaces, this may be used to bind the coordinator port to a specific interface. The host name must be a valid host name on the local machine.
-v	Normally all output to the console from engines is redirected to a log file in the run time directory called coordinator.stdout. If this option is specified, all console ouput will go to the console instead. Please note that output to coordinator.stdout is buffered, so output from engines may not be immediately visible in this file until either (a) the output buffer is filled and is flushed or (b) the engine process terminates, which also flushes the output buffer.
-W	Set a new node password. If a password already exists, -w must be used to also specify the old password. Once a password is set, -w will be required to all subsequent swsrv commands.

Your path should be set correctly before you use KIS commands. See “User Environment” on page 3

System engines

`swsrv` automatically starts all of the engines in the System group:

Java Name Server	name service for Java clients
swcoordadmin	node administration
osw	PHP access to node
kssl	security features
swadminui	statistics

Do not stop any of the system components. The system components are necessary for the KIS node to function properly. Stopping a system component could force the node to go through recovery.

Host and Port Numbers

When you start a KIS node with `swsrv` , it starts up network listeners for the following services:

- System Coordinator
- Name Server
- Web Server

For each of these network listeners it is possible to specify command line options to control the host and port numbers on which the listener is started.

For example, it is possible to specify that the network host name is “ `localhost` ”. `localhost` indicates that only connections from the local machine can be used to access the network listener. By default the system coordinator and Web Server listen on all network interfaces installed in the machine. This is indicated by the `swsrv` startup messages of:

```
System Coordinator Host: All Interfaces
System Coordinator Port: 1031

Web Server Host: All Interfaces
Web Server Port: 1033
```

If one of these services is started on a specific host, the `swsrv` start-up message will look like:

```
Name Server Host: plato
Name Server Port: 1032
```

See the table above for the specific command-line options that control the host and port configuration

Node configuration

Each KIS node has global configuration information that affects all engines and components running on the node. Node configuration variables can only be configured through deploy specifications and the ECC (see Chapter 3) .

3

Managing Engines

This chapter tells you how to manage engines and components installed and running on a Kabira Infrastructure Server (KIS) node.

KIS applications are comprised of one or more components running as a set of processes known as engines on a node.

A component is the unit of reuse and deployment. All components have a public interface that defines the services the component provides. Normally, you use the Design Center to generate components from models (see Creating ObjectSwitch Applications); however you can implement components using other technologies as well. From the outside, it is both impossible and unimportant to know how the component is implemented.

You assign components to one or more processes, known as engines, that execute on a node. A node is the application service environment for components. The node provides all the runtime services and management infrastructure necessary to execute components (On managing nodes, see Chapter 2).

You use a declarative deploy specification to assign components to engines and configure those engines. Deploy specifications are like installation scripts; they identify the components that make up an application, they specify how those components are assigned to processes, and they define any runtime properties that require special configuration.

Deploy specifications are loaded into a node using either the command line tool `swnode` , or the graphical tool Engine Control Center.

This chapter contains the following sections:

- the section called “ Engines ” on page 12
- the section called “ Deploy specifications ” on page 13
- the section called “ Security management ” on page 19

- the section called “ Distributed discovery management ” on page 23
- the section called “ Engine management tools ” on page 23
- the section called “ Adding, replacing, and removing engines ” on page 28
- the section called “ Viewing engines on a node ” on page 31
- the section called “ Starting and stopping engines ” on page 34
- the section called “ Changing an engine’s configuration ” on page 35
- the section called “ Generic engine properties ” on page 37
- the section called “ Trace files ” on page 40
- the section called “ Engine lifecycle events ” on page 41

Engines

Engines are the unit of execution for KIS. Each engine maps directly to an operating system process. An engine can host one or more components.

Generally speaking, it is best to combine as many components in the same engine for the best possible performance. This allows the runtime to dispatch events via direct function calls rather than across the event bus to a different process.

However, there are valid reasons why you would not want to combine components in the same engine, for example, when a component:

- makes heavy use of process resources (such as file descriptors) and needs to run separately
- must start and stop independently of other components
- specifically forbids being run with other components in the same process (see the section called “ componentType ” on page 13 and the section called “ componentSingleton ” on page 13)

You manage engines (start, stop, etc.) with the graphical and the command-line engine management tools (see the section called “ Engine management tools ” on page 23).

There are many generic engine properties that you may configure. These include path names, executable type, log files, recovery policies, and tracing information, among others (see the section called “ Changing an engine’s configuration ” on page 35).

In addition, components may add specific configuration properties to the engines in which they reside. For example, JSA-enabled components add Java specific properties. Fortunately, all of these properties have sensible defaults and most components will run with no extra configuration. Where specific configuration is required, you can specify those properties in the deploy specification along with the engine definitions (see the section called “ Deploy specifications ” on page 13).

Once you install an engine, you can view and modify all of its properties with one of the engine management tools.

Components

Building components from models is described in detail in Creating ObjectSwitch Applications. You define component properties in the component specification used to build the component. Generally, component properties are only of interest at build time. However, there are a few components properties that directly affect component deployment:

componentType The following table shows different possibilities for componentType and describes deployment aspects:

DYNAMIC	This is the normal component type for KIS components. This property says that the component can be dynamically loaded into an engine and can co-exist with other DYNAMIC components.
STATIC	This is a special case, which says the component cannot be dynamically loaded. Typically this happens when a component must link with third party libraries that do not support dynamic loading. When this property is specified, the Design Center builds a statically linked component that must be the only STATIC or DYNAMIC component assigned to an engine.
LIBRARY	LIBRARY types provide support functions such as the swbuiltin package. They may be assigned to any engine that contains components that require these functions, both STATIC and DYNAMIC.

componentSingleton This property indicates that only one instance of a component can be running on a node. Typically, you want to specify a singleton component if the component stores process resources in attributes and cannot support multiple instances. It is non-deterministic which component instance services an event. Thus, the situation can arise where a process resource created in one engine is handled by a different engine where that process resource is not valid.

Deploy specifications

Deploy specifications define

- engines
- engine properties
- node properties.
- components assigned to engines
- search paths

Creating a deploy specification

The format of deploy specifications is similar to component specifications. Each deploy specification starts with a deploy keyword:

```
deploy MyApp
```

```
{
   ...
};
```

The deploy keyword serves several purposes:

- it groups together all engines defined within a deploy block together in the ECC
- it scopes the names of engines to the deploy block; for example, if the deploy block is called `MyApp` , then the names of all engines in the block start with `MyApp::` , both for `swnode` and the ECC.
- it allows you to group sets of components together for reuse (see the section called " Exporting and using components and properties " on page 17); this is useful when shipping component frameworks

Defining engines

Within a deploy block, you define engines, with the engine keyword. You can define any number of engines:

```
deploy MyApp
{
    engine EngineA
    {
    };
    engine EngineB
    {
    };
};
```

As stated, engines are scoped to the deploy block, so the fully scoped name of `EngineA` is `MyApp::EngineA` in the engine management tools.

Adding components to engines

Within an engine block, you list the components that you want to assign to the engine with the component keyword:

```
deploy MyApp
{
    engine EngineA
    {
        component Comp1;
        component Comp2;
    };
    engine EngineB
    {
        component Comp1;
        component Comp3;
    };
};
```

When you load a deploy specification, the engine management tools ensures that the deploy specification does not violate any component constraints.

For example, they ensure that a DYNAMIC component has not been placed in the same engine as a STATIC one or that a singleton component does not appear in more than one engine.

Setting component search rules

There are two default search paths where the engine management tools look for components: " `.` " and `$SW_HOME/distrib/$SW_PLATFORM/component` . You can provide additional search paths with the importPath property.

```
importPath=/search/path/one;
deploy MyApp
{
    importPath=/search/path/two;
    engine EngineA
    {
        importPath=/search/path/three;
        component Comp1
        {
            importPath=/search/path/four;
        };
    };
};
```

Search paths are processed from innermost scope outward. In the previous example, the search order for `Comp1` would be:

```
/search/path/four
/search/path/three
/search/path/two
/search/path/one
```

When you specify a component that you want to add to an engine, you do not supply a path or file extension.

Setting engine properties

You can provide additional engine configuration properties to override defaults. For example:

```
deploy MyApp
{
    engine EngineA
    {
        buildType=DEVELOPMENT;
        component Comp1;
    };
};
```

You can scope some engine properties to their own blocks. To do this, use the config keyword to specify the block:

```
deploy MyApp
{
    engine EngineA
    {
        config Environment
        {
            envVar=someValue;
        }
        config Java
        {
            classPath=/some/path;
        };
        component Comp1;
    };
};
```

The engine property must be valid; otherwise a load error occurs. See the section called " Generic engine properties " on page 37 for a list of the available properties.

Setting node properties

You can also set node properties. Use the config keyword to specify a block within the deploy block:

```
deploy MySecureApp
{
    config Security
    {
        config Principals
        {
            config "MyPkg::MyEntity"
            {
                textCredential="shh";
                Roles=$ROLE1 $ROLE2 $ROLE3;
            };
        };
    };
    engine EngineA
    {
        component Comp1;
    };
};
```

See the section called " Generic engine properties " on page 37.

Using special syntax

You can use either the C or C++ styles for comments:

```
/*  C style
    comment */
// C++ style comment
```

You can also quote special characters or keywords in that appear in properties:

```
deploy MyApp
{
    engine EngineA
    {
        config "config"
        {
            "-arg"="some value";
        }
        component Comp1;
    };
};
```

Strings can span multiple lines, or split into parts to avoid the newline:

```
deploy MyApp
{
    engine EngineA
    {
        config "config"
        {
            "-arg"="some"
            " value";
            string="this is
                    a multiline
                    string";
        }
        component Comp1;
    };
};
```

Strings can also contain the following special sequences: `\n` , `\t` , `\v` , `\b` , `\r` , `\f` , `\a` , and `\\` .

Using macros

Deploy specifications support macros. Macros substitution takes place before the file is parsed, so you can use macros for anything. Typically, you use them for property values. For example:

```
deploy MyApp
{
    engine EngineA
    {
        buildType=$(BUILD);
        component Comp1;
    };
};
```

You specify the values for macros on the command line to swnode. There is currently no way to specify macros from the ECC.

Exporting and using components and properties

You may want to ship products as frameworks, designed for use by other programmers. Deploy specifications provide a syntax that allows you to specify a framework as a single unit.

The export keyword groups a set of components and properties that define a framework:

```
deploy MyFramework
{
    export
    {
        property=value;
        component Comp1;
        component Comp2;
    };
};
```

You can then refer to the framework from another deploy specification:

```
importPath=/some/path;
import frmwk;
deploy MyApp
{
    engine EngineA
    {
        use MyFramework;
        component Comp3;
    };
};
```

Typically this is a two step process:

- import the framework's deploy specification
- specify engines that use properties and components that the framework exports

The import statement allows you to insert the contents of the specified deploy specification at the point of the import statement. You specify the name of the file without path and file extension. The system searches for the `.kds` file using the same search path as components. You can use importPath statements to augment the search path, but they must appear before the import statement in the file.

You specify that an engine uses exported properties and components with the use statement. The use statement expects the name of a deploy specification. The system inserts the contents of all export blocks found in the specified deploy specification into the engine that uses it. This includes properties set within the export block. If the engine block sets the same property, the engine block's property setting takes precedence.

There can be multiple export blocks in a deploy block.

The following example illustrates how the system processes import, export, and use statements. Assume there are the following two deploy specifications, in separate files, `frmwrk.kds` and `client.kds` .

File `frmwrk.kds` :

```
deploy MyFramework
{
    export
    {
        name=value1;
        component Comp1;
    };
    engine EngineA
    {
        component Comp2;
    };
};
```

File `client.kds` :

```
import frmwrk;
deploy MyApp
{
    engine EngineB
    {
        use MyFramework;
        component Comp3;
    };
};
```

Loading `client.kds` results in a deploy specification that you could also write as follows:

```
deploy MyFramework
```

```
{
    engine EngineA
    {
        component Comp2;
    };
}
deploy MyApp
{
    engine EngineB
    {
        name=value1;
        component Comp1;
        component Comp3;
    };
};
```

Adding an external engine

You can define an engine that allows the System Coordinator to control the execution of external programs.

Use a deploy specification like the following to create an external engine:

```
deploy YourExternalEngine
{
   engine YourExternalEngine
   {
      component ExternalProgram;
      // Required configuration
      name="Display Name";
      description="Description";
      developmentExecutable=/path/to/executable;
      productionExecutable=/path/to/executable;
      engineArguments="-arg1 -arg2";
      // Optional configuration
      autoStart=(true false);
      restart=(true false);
      needsRecovery=(true false);
      triggersRecovery=(true false);
      maxFailCount=10;
      engineExitValues="0";
      engineKillSignal=15;
      config Environment
      {
      };
   };
};
```

Security management

You can define the security policy for model elements. The security policy lets you specify who can access elements in a component, and what type of access is allowed on each of those elements. You specify this policy using directives within a deploy specification (see the section called “ Deploy specifications ” on page 13), as described in the following paragraphs. You can also remove security directives using a deploy specification, as described in the section called “ Removing configuration entries ” on page 30.

Security policy concepts and features

A security policy specifies the access control rules that apply to roles associated with authenticated principals. These concepts are described below.

Authentication and principals A principal is a user or agent on whose behalf work is to be done in the system. The process of authentication involves verifying the identity of a principal accessing a specific node.

Secured model elements can only be accessed using adapters that provide authentication. One example is the authenticated form of os_connect() in the osw PHP engine.

Principals and roles An authenticated principal can have one or more roles associated with it. A role describes a user type, such as “customer” or “administrator”.

Roles and access control Access to elements in the component is governed by access control rules. A rule grants permission for a role to access an element of the component, and specifies the type of access allowed.

Access permission is always granted to a role, and thus all principals associated with that role. Explicit access permission cannot be granted directly to principals.

Configuring security

KSSL lets you specify access control rules for the following model elements:

- Interfaces
- Individual attributes and operations within an interface
- Relationships defined for interfaces
- Extents of interfaces—access by select
- Keys with interfaces—access by select ... where <key=value>

You can also specify access control rules on model elements at a higher scope:

- Module—applies to all interfaces in the module
- Package—applies to all interfaces in the package

Security syntax

You can add security directives to your deployment specification to define the following policy elements:

- Principals, with their associated roles and text credentials
- Access control rules

Each of these types of directives forms a section within a deploy specification.

Principals, roles, and credentials You configure principals, together with their roles and credentials, in the Principals section of the deploy specification. A principal definition applies to an entire KIS node. The syntax is as follows:

```
config Principals
{
    config principalname
    {
        textCredential=password;
        roles=role_set;
    };
    . . .
};
```

The principal definition may include any of thedefinitions listed in the following table.

name	type	default	description
deferredCredential	boolean	false	Credential supplied on first authentication
textCredential	string		Text Credential
roles	string		Roles for principal
credentialExpirationPeriod	integer	0	Number of days that the credentials remain valid
trusted	boolean	false	Principal can authenticate from trusted hosts based only on the host’s own authentication of the principal.
credentialRequired	boolean	fa;se	Principal must present a credential regardless of the value of the “trusted” definition.

Access control rules Access control rules are applied to model elements in the AccessControl section of the deploy specification. The syntax is:

```
config AccessControl
{
    config model_element
    {
         locked=true;
        role_name=permission_set;
    };
};
```

Where permission_set is one or more of the following, separated by the character:

- `Read`
- `Write`
- `Execute`
- `Create`
- `Destroy`

- `Select`
- `Relate`

For example, a rule that grants Select and Create access to the “admin” role would look like:

```
admin=Select Create;
```

The `locked` keyword is used to disable all access to the element except that which is granted by the assignments that follow. You can also lock all elements in an interface with a single lockAllElements directive:

```
config Security
{
    config AccessControl
    {
        config interface-scoped-name
        {
            lockAllElements=true;
        };
    };
};
```

Different types of access may be granted on different element types. The following table describes which permissions apply to each type of model element:

	Interface	**Attribute**	**Operation**	**Role**
Create	X			
Destroy	X			
Read		X		
Write		X		
Select	X			X
Relate				X
Execute			X	

Example

The following example shows the format and syntax of security policy directives in a deployment specification:

```
// mydeployspec.kds
deploy myNode
{
    config Security
    {
        config Principals
        {
            // "SomeUser" is our principal name
            config SomeUser
            {
                textCredential=somePassword;
                // a set of roles assigned to the principal
                // each role separated by " "
                roles=customer stockHolder;
            };
        };
        config AccessControl
        {
            config mypackage::myinterface
            {
                // block all access expect as granted below
                locked=true;
                // new grant specific access to myinterface
                config Permissions
                {
                        customer=Select;
                        stockHolder=Create Select Relate Destroy;
                };
            };
        };
    };
};
```

Distributed discovery management

Distributed applications may use a distributed discovery protocol to identify nodes participating in the application. On multi-homed hosts it is necessary to specify which network interface to use for the discovery protocol.

You specify this in the distribution section of a deployment specification as follows:

```
config Distribution
{
    config DiscoveryService
    {
        broadcastHost=hostName;
    };
};
```

Where hostName is the host name of the interface that the distribution engine will use for discovery service. The default hostname is the name returned from the gethostname(3C) library call.

Hosts that are not multi-homed do not need a distribution section in their deployment specifications.

Engine management tools

KIS provides a graphical and command-line tool for managing engines on a node: the Engine Control Center (ECC) and `swnode` , respectively.

To manage engines through a graphical user interface, start the ECC with the command `swecc` .

To manage engines from the command line, use the `swnode` command-line tool.

Both commands need to know which node to connect to. Use one of the flags in the following table:

Parameter	Description
-p	Specify the path to the node runtime directory. This is useful only if the coordinator is running on the same physical host.

Parameter	Description
-P	Specify the node coordinator port. Be sure to choose the coordinator port and not one of the other ports.

You can start the engine management tools without either the `p` or `P` option and still connect to a node, if you invoke the respective command in the node runtime directory.

In addition, the following parameters may be required:

Flag	Description
-w	Specify the node password. This must be provided if a password has been configured for the node (see swsrv -W).
-h	Specify the node host. This is used if the node resides on a different physical host. In this case, the -P option must be specified as well.
-t	Specify a coordinator response timeout in seconds. If swnode is used from scripts, it may be desirable to set a timeout so that swnode doesn't wait forever for a coordinator response. However, this timeout must be long enough for the longest command that can be executed. For example, the "load" command (which loads deploy specifications) can take a long time. Normally there is no timeout and the swnode command will wait forever.
-o name=value	Specify a values for macros in deploy specifications. For example: `swnode -o BuildType=DEVELOPMENT` This will result in "DEVELOPMENT" being substituted for each $(BuildType) macro.

Engine Control Center

The Engine Control Center lets you control engines on any node. You can use it to start, stop, add, and remove components and engines, and to configure settings associated with individual engines. With the administration engine running you can also get performance statistics and assistance upgrading engines.

You start the Engine Control Center from a shell prompt with the following command:

```
swecc [-h host ] [-P port ]
```

Your path should be set correctly before you use KIS commands. See “User Environment” on page 3.

The `-h` host and `-P` port arguments are optional. By default, the ECC attempts to contact a System Coordinator listening on port 8387 (the default coordinator port) on the local machine. The port on which the System Coordinator listens is set when you start the coordinator using the `swsrv` command. You can also open other System Coordinators once you have opened the ECC.

When the ECC starts, it attaches to the System Coordinator. Figure 3.1 shows the display of the Engine Control Center when it is attached to a System Coordinator running on port 31335 on the localhost.

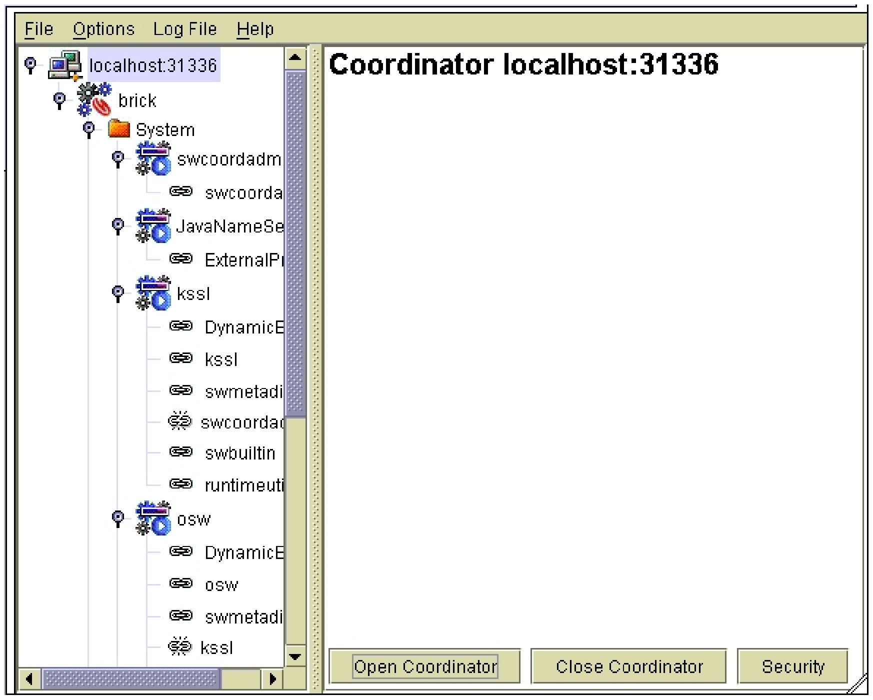
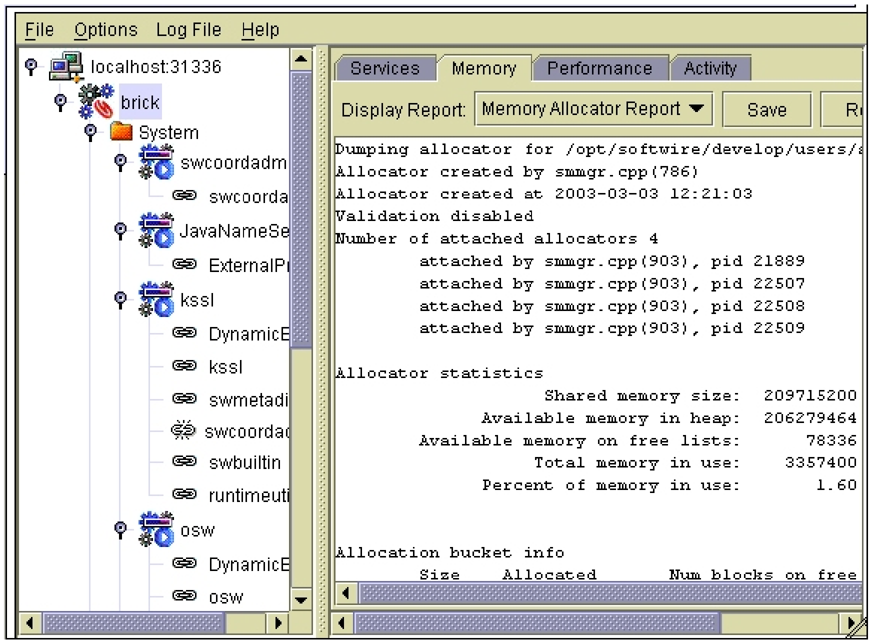


Figure 3.1. The Engine Control Center

You can start the Engine Control Center using the port and server arguments or start it without these arguments and then attach to a coordinator by clicking on the Open Coordinator button. In either case, the Engine Control Center connects to the System Coordinator for the node you specify. The left-hand pane lets you browse through a hierarchy of hosts, coordinators, engine groups, and engines. A host can have several coordinators running on it, a coordinator has one or two engine groups, and an engine group can have many engines.

Statistics reports The Engine Control Center lets you create a variety of statistics reports to help diagnose system behavior. Select a node and click on Statistics to display the statistics window, shown in Figure 3.2.



```
Dumping allocator for /opt/softwire/develop/users/
Allocator created by smmgr.cpp(786)
Allocator created at 2003-03-03 12:21:03
Validation disabled
Number of attached allocators 4
        attached by smmgr.cpp(903), pid 21889
        attached by smmgr.cpp(903), pid 22507
        attached by smmgr.cpp(903), pid 22508
        attached by smmgr.cpp(903), pid 22509


Allocator statistics
                   Shared memory size:  209715200
             Available memory in heap:  206279464
        Available memory on free lists:     78336
                   Total memory in use:   3357400
              Percent of memory in use:      1.60


Allocation bucket info
        Size    Allocated       Num blocks on free
```

Figure 3.2. The statistics window

The statistics window contains a list of report types and queue IDs. The following reports collect data over time:

- Execution Statistics
- Transaction Statistics
- Mutex Statistics

Data collection for these three reports is disabled by default (data collection is detrimental to system performance); you must enable it for the reports to be meaningful. All other reports reflect a static snapshot of shared memory.

When you create a report, it displays in the lower part of the statistics window. You can save the report to a file by clicking on the Save button.

Report	**Description**
Execution Statistics	Performance statistics including number of deadlocks, exceptions, and min/avg/max execution times. Use this report to locate slow operations in your model.
Transaction Statistics	List of read and write locks and contention, number of deadlocks, and lock acquire times at the transaction level. Use this report to identify resources involved in excessive lock contention.
Event Bus Statistics	General technical information about the event bus. You will probably not use this report.
Queue ID Listing	A list of the queues shown at the right in Figure 3.2. Use this report if you need to save or print a list of the queues.

Report	Description
Queue Report	Information about the status of the selected queue and events currently on that queue. Use this report to see the events that are queued for an engine.
Subscription List	A list of the event subscriptions for the selected queue. You generally do not need to use this report.
Mutex Statistics	This report describes low-level mutex locks, contention, and acquire times. This is a very technical report and not easy to understand without a very strong knowledge of KIS internals. You will probably never use this report.
Memory Allocator Report	Size and content of type hashes. This is a very technical report and not easy to understand without a strong knowledge of KIS internals. You will probably never use this report.

Log File Browser A log file browser is provided by the Engine Control Center. This provides a mechanism to remotely monitor log files associated with engines running on a node.

To access the log file browser click on the Log File button on the engine content panel. The ECC reads the log file data from the coordinator on the remote node. The log file data is parsed and displayed as a table.

The Log File menu has a list of controls to customize the table display. Specific columns can be hidden and optionally wrapped to the column and row height. It is also possible to color code the different trace kinds using the Preferences menu.

The log data can be refreshed using the Log File->Refresh menu item. This will cause any updates to the log file on the remote node to be read and displayed in the ECC.

It is possible to jump to the end of a displayed log by clicking on the Down button at the top right corner of the displayed table.

swnode

`swnode` has an additional command-line flag that you can use to specify values for macros:

Parameter	Description
-o name=value	Specify a values for macros in deploy specifications. For example: `swnode -o BuildType=DEVELOPMENT` This will result in "DEVELOPMENT" being substituted for each $(BuildType) macro.

`swnode` can process its command input from three sources:

Command-line argument For example:

```
swnode -p runpath getengines
```

Standard input For example:

```
swnode -p runpath <<EOF
  getengines
  startnode
EOF
```

Command file For example:

```
swnode -p runpath -f commandfile
```

Once a node is running, you can use `swnode` to load deploy specifications and control engines.

Adding, replacing, and removing engines

You deploy an engine by loading the corresponding deploy specification (see the section called " Deploy specifications " on page 13). You can load a deploy specification with either of the engine management tools.

When you deploy an engine for the first time, the coordinator sets the engine's configuration to the default values built into the component archive or those overridden in the deploy specification. It creates a deployment directory for the engine, a subdirectory of the node's runtime path. It extracts all the files necessary for deploying the component from the component archive to the deployment directory.

For dynamic engines (engines containing dynamic components), a load or replace causes a reload to occur. This means that the coordinator unloads all components and then reloads those specified in the deploy specification. This in turn invokes any operations in the model with the [unload] and [reload] properties. All engine settings are retained.

Reloading or replacing a static engine (an engine containing a static component), or changing an engine from dynamic to static (or vice versa), stops and restarts the engine. All engine settings are retained.

When you reload a deploy specification, the components may or may not be compatible with the existing components (see the section called " Type mismatch errors " on page 30). Changes to a component that are compatible with earlier versions include:

- adding new entities and interfaces
- adding triggers
- changes to action language

Incompatible changes to an engine include:

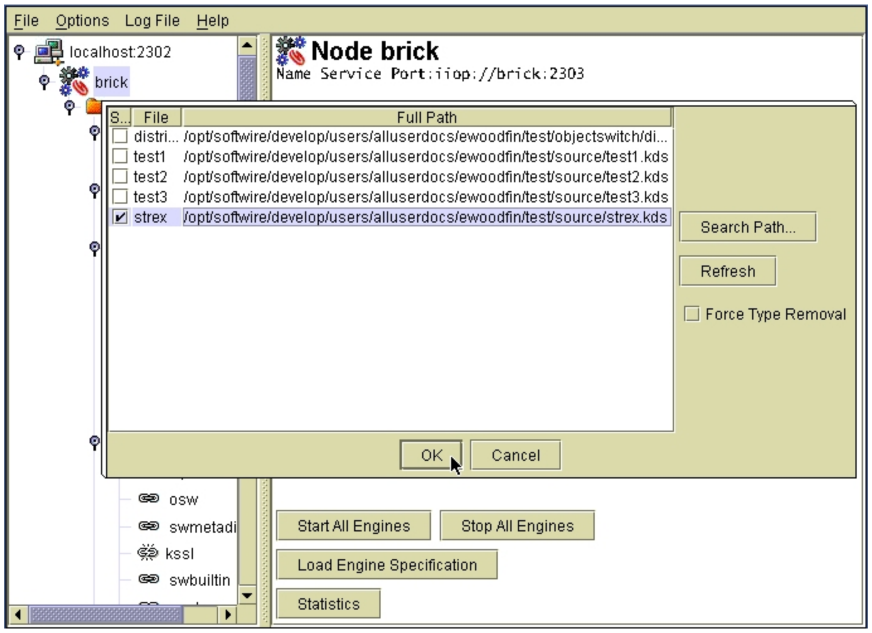
- adding or removing attributes
- changing operation signatures
- adding or removing states or transitions

ECC

From the ECC you can add, replace, and remove engines.

Adding an engine Click on the node where you want to deploy the engine, and then click on the Load Engine Specification button. This brings up a dialog that allows you to select any of the deploy specification files found in one of the search paths. This same dialog allows you to add a new search path by clicking on the Search Path ... button.

Click on the OK button to load the selected deploy specifications.



S...	File	Full Path
☐	distri...	/opt/softwire/develop/users/alluserdocs/ewoodfin/test/objectswitch/di...
☐	test1	/opt/softwire/develop/users/alluserdocs/ewoodfin/test/source/test1.kds
☐	test2	/opt/softwire/develop/users/alluserdocs/ewoodfin/test/source/test2.kds
☐	test3	/opt/softwire/develop/users/alluserdocs/ewoodfin/test/source/test3.kds
✓	strex	/opt/softwire/develop/users/alluserdocs/ewoodfin/test/source/strex.kds

Figure 3.3. Selecting a deploy specification

When you deploy a component, it appears in the ECC and is not started.

Replacing an engine When you replace an engine, you are adding the engine and removing any conflicting types. To replace an engine, you go through the same steps as adding an engine, but in the dialog for selecting the engine deploy specification file, select the Force Type Removal check box before clicking on the OK button. Use with caution (see the section called "Type mismatch errors " on page 30).

Removing an engine Click on the engine and then click on the Remove button. The ECC will not let you remove an engine that is running; you must stop it first.

swnode

swnode lets you add, replace and remove engines.

Adding an engine The `load` command loads the specified file as a engine specification.

```
swnode load <filename>
```

Replacing an engine The `replace` command is identical to the `load` command except that it applies the Force Type Removal option. Use with caution (see the section called " Type mismatch errors " on page 30).

```
swnode replace <filename>
```

Removing an engine The `remove` command removes the specified engine.

```
swnode remove <engine>
```

Removing configuration entries The `unload` command removes the configuration items specified in a deploy specification.

```
swnode unload <filename>
```

The contents of the deploy specification are processed to remove any engine definitions and application-specific configuration values. Security policy directives are de-activated as follows:

- any defined principals are removed
- any defined trusted hosts are removed
- any defined access control rules are removed
- any defined audit directives are removed

Type mismatch errors

If you receive a type mismatch error when loading a deploy specification, components being added contain types that conflict with types already in shared memory. You cannot allow this situation to persist; you must remove the corresponding engine.

You can replace an engine with the Force Type Removal option; however, use this with extreme caution, as as it removes from shared memory all instances of a type and all events destined for it. Furthermore, it is not a panacea for type mismatch errors; type mismatches will continue to occur if you do not remove the conflicting engines.

If you encounter a type mismatch error, proceed as follows:

1. determine which component contains the object types that conflict (with dynamic engines, the same component may appear in multiple engines; you must reload each one of these engines with the new component, or remove altogether)
2. stop all engines.
3. remove any engines that should no longer exist in the system
4. load the deploy specification for the affected engines using the Force Type Removal option
5. start all engines.

Viewing engines on a node

KIS allows you to view engines, the assigned components, engine status, engine properties, and dependencies.

ECC

You can then navigate through the hierarchy "tree" to display the node's engines and their status.

When you first open a node using the ECC you'll see the System engine group, which contains the following engines:

- an administration engine `swcoordadmin`
- a name server engine `JavaNameServer`
- a web engine `osw`
- a security engine `kssl`
- a statistics engine `swadminui`

`swsrv` starts these engines automatically when you start a node (see "Node command-line tool" on page 7).

`swsrv` starts the naming service engine using a random free port or the port you specify with the n option to the `swsrv` command. If you specify a port, `swsrv` uses that port, whether it is free or not.

The administration engine `swcoordadmin` is a special engine that handles most administrative tasks such as engine install, removal, and configuration; you cannot modify or manage this engine.

The web engine `osw` enables access from the Internet to KIS engines with interfaces exposed through the PHP adapter.

The `swadminui` engine handles statistics.

The `kssl` engine handles security.

The ECC groups engines according to the deploy block names used in the corresponding deploy specification.

For example, assume you load the following deploy specification:

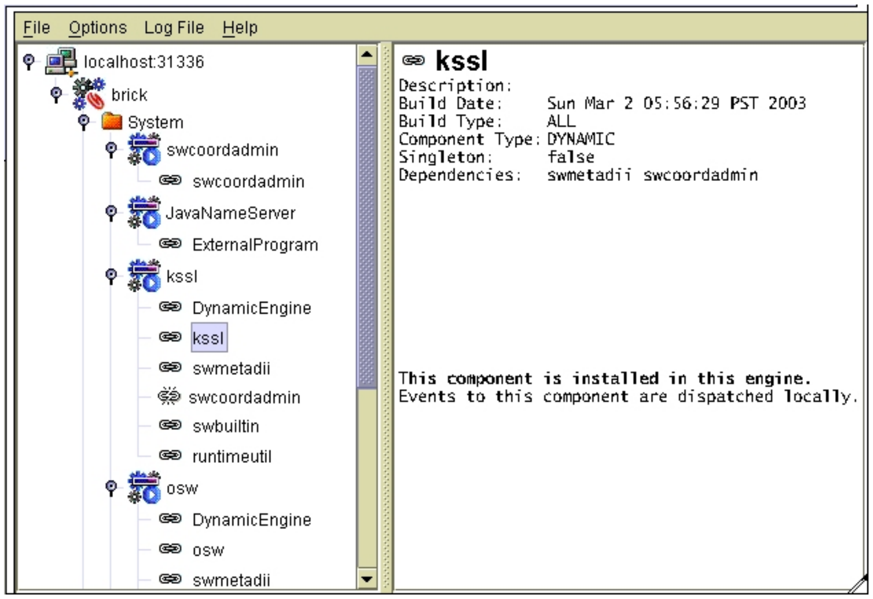
```
deploy MyApplication
{
   engine MyEngine
   {
      ...
   };
};
```

The ECC would place the engine `MyEngine` in the `MyApplication` folder.

When you first connect to a node using the ECC, all the system engines are visible (on managing nodes, see Chapter 2). The icons show that the engines are running, and by clicking on an engine's icon, you can display a more detailed status. The components of an engine appear under the engine icon. Clicking on a component icon displays more detailed status.

This display shows both the components in the engine and the components upon which this engine depends and which must be installed elsewhere on the node. The icons are visually different (see the following table).

Figure 3.4 shows how the detailed status is displayed for the `kssl` component in the `kssl` engine.



```
kssl
Description:
Build Date:     Sun Mar 2 05:56:29 PST 2003
Build Type:     ALL
Component Type: DYNAMIC
Singleton:      false
Dependencies:   swmetadii swcoordadmin

This component is installed in this engine.
Events to this component are dispatched locally.
```

Figure 3.4. Displaying an engine's status

The Engine Control Center displays hosts, nodes, engine groups, and each engine attached to a node, as well as the components assigned to each engine, using an icon that represents these elements. There are also several different engine icons to reflect an engine's status.

The following table describes these icons.

Icon	Description
🖥	A host. When the Engine Control Center connects to a coordinator, the host is displayed at the top level of the hierarchy.
⚙	A KIS node. There can be one or more nodes that appear under a host.
📁	An engine group. There can be one or more engine groups that appear under a KIS node.
⚙	An engine that is not running. There can be one or more engines that appear under an engine group.
⚙	An engine that is running.

Icon	Description
⚙	An engine that is stopping. This icon appears briefly when you stop an engine, but it is not yet idle.
🔗	A component assigned to the engine.
🔗	A dependent component not assigned to the engine.

You can view several nodes at once. To open another node, click on the host and then click on the Open Node button. A dialog appears to let you select a host and port.

swnode

With swnode you can query a node for

- installed engines
- assigned components
- engine properties
- engine status
- process IDs
- dependencies

Listing engines The `getengines` command returns a list of engines installed on the node.

```
swnode getengines
```

Querying for the state of an engine The `getstate` command returns the state of the specified engine. This indicates whether the engine is running, stopped, recovering, etc.

```
swnode getstate <engine>
```

Query for the process ID of an engine The `getpid` command returns the process ID of the engine if it is running.

```
swnode getpid <engine>
```

Query for the process exit value The `getexitcode` command returns the process exit value of the engine if it is stopped.

```
swnode getexitcode <engine>
```

Returns the process exit value of the engine if it is stopped.

Listing dependencies The `getdeps` command returns the list of components required by, but not implemented in, this engine. These components must exist in some other engine to satisfy the dependencies.

```
swnode getdeps <engine>
```

Listing components The `getcomps` command returns the list of components implemented in this engine.

```
swnode getcomps <engine>
```

Listing unassigned components The `getunassigned` command returns the list of component dependencies for an engine that are installed on the node, but are not assigned to an engine.

```
swnode getunassigned <engine>
```

Listing component dependencies The `getmissing` command returns the list of component dependencies for an engine that are not installed on the node.

```
swnode getmissing <engine>
```

Starting and stopping engines

You should not attempt to start or stop the engines in the System group. Other than this, you can start and stop engines in any order.

ECC

When you click on an engine as shown in Figure 3.4, the Engine Control Center displays the status of that engine. There are also buttons to start, stop, configure, remove, and view the engine's log file. Click on Start to start the engine. Click on Stop to stop a running engine.

You can also start or stop all the engines in an engine group by selecting the engine group and then clicking on the Start All Engines button or the Stop All Engines button.

If you use the engine control center to shut down an engine, the coordinator won't restart it—even if it is configured to autostart—until you start it again manually. This lets you shut down engines for maintenance without changing their auto-start configuration.

swnode

The swnode tool allows you to:

- start engines
- stop engines
- restart engines

Starting all engines The `startnode` command starts all engines on the node.

```
swnode startnode
```

Stopping all engines The `stopnode` command stops all engines on the node.

```
swnode stopnode
```

Starting one engine The `start` command starts the specified engine.

```
swnode start <engine>
```

Stopping one engine The `stop` command stops the specified engine.

```
swnode stop <engine>
```

Restarting an engine The `restart` command restarts a dynamic engine. It reloads the deploy specification. A restart takes place automatically whenever you load a deploy specification.

```
swnode restart <engine>
```

Changing an engine's configuration

There are many properties that you can configure for engines, for example:

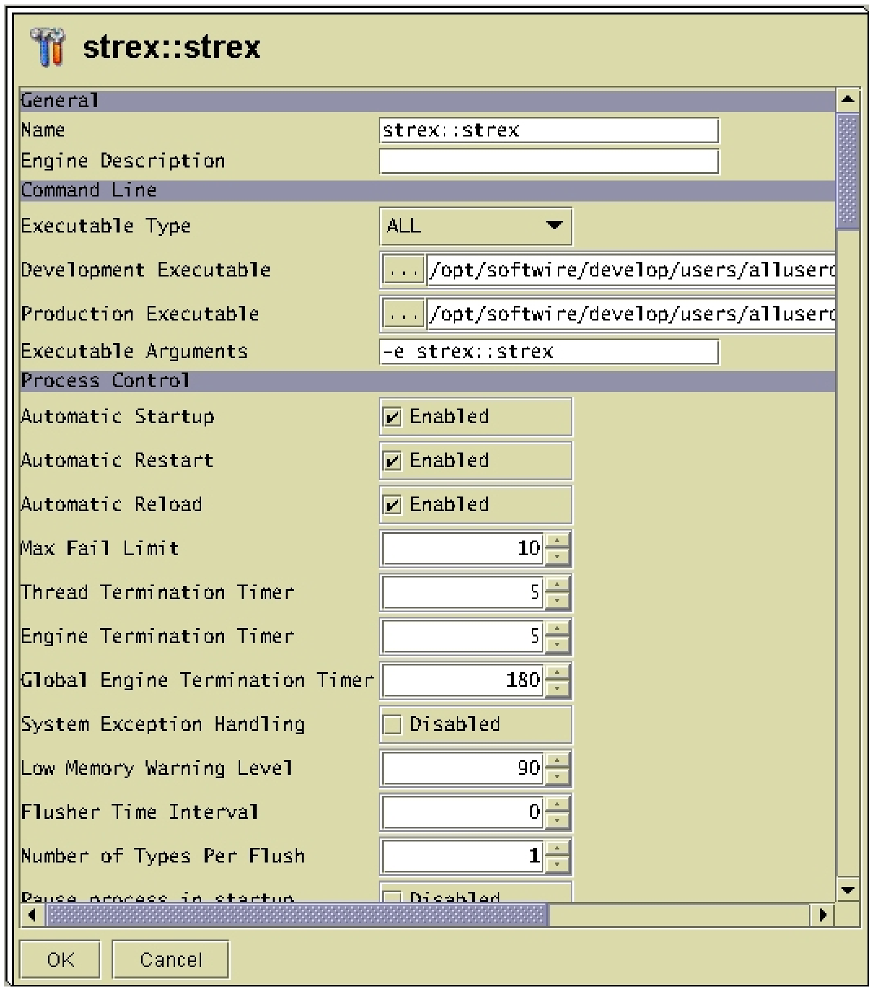
- path names
- executable type
- log files
- recovery policies
- tracing information

Additionally, components may add specific configuration properties to the engines in which they reside; for example, JSA-enabled components add Java specific properties. Fortunately, these properties have sensible defaults and most components run without extra configuration.

You can configure an engine through its deploy specification (see the section called " Deploy specifications " on page 13). Or once you install an engine, you can view and modify all properties with one of the engine management tools.

ECC

You can edit an engine's configuration by selecting it and then clicking on the Configure button. This displays the configuration panel for the engine as shown in Figure 3.5. You can't configure the `JavaNameServer` engine.



General	
Name	strex::strex
Engine Description	
Command Line	
Executable Type	ALL
Development Executable	/opt/softwire/develop/users/alluser
Production Executable	/opt/softwire/develop/users/alluser
Executable Arguments	-e strex::strex
Process Control	
Automatic Startup	✓ Enabled
Automatic Restart	✓ Enabled
Automatic Reload	✓ Enabled
Max Fail Limit	10
Thread Termination Timer	5
Engine Termination Timer	5
Global Engine Termination Timer	180
System Exception Handling	☐ Disabled
Low Memory Warning Level	90
Flusher Time Interval	0
Number of Types Per Flush	1
Pause process in startup	☐ Disabled

Figure 3.5. Engine configuration properties

You can edit properties in this window, then click on OK to accept your changes. You can click on Cancel to discard your changes.

The properties are described through tooltips that pop up when you pass the mouse over the property label in the dialog. The properties are grouped into sections in the Engine Control Center; and are described below under those same headings (see the section called " Generic engine properties " on page 37).

Engines that use adapters may have additional sections with properties that apply to the adapters in the engine. These properties are described in the relevant configuration sections of Using Adapters.

swnode

The swnode tool allows you to configure engines in the following ways.

Setting an engine or node property The `setparam` command sets one engine property. If the engine name is omitted, it sets a corresponding node property.

```
swnode setparam [<name>] <param>=<value> [<param>=<value> ...]
```

Examples:

Set the node property `SystemCoordinator/tcpPort`

```
swnode setparam SystemCoordinator/tcpPort=8387
```

Set the engine property `buildType` for `MyGroup::MyEngine` .

```
swnode setparam MyGroup::MyEngine buildType=PRODUCTION
```

Querying for an engine property The `getparam` command returns an engine or node property. If the engine name is omitted, it returns a node property.

```
swnode getparam [<name>] <param>
```

Examples:

Return the node property `SystemCoordinator/tcpPort`

```
swnode getparam SystemCoordinator/tcpPort
```

Return the engine property `buildType` for `MyGroup::MyEngine` .

```
swnode getparam MyGroup::MyEngine buildType
```

Removing an engine or node property The `unsetparam` command removes an engine or node property. If engine name is omitted, removes a node property.

```
swnode unsetparam [<name>] <param>
```

Example:

Remove the node property `SystemCoordinator/tcpPort` .

```
swnode unsetparam SystemCoordinator/tcpPort
```

Remove the engine property `buildType` for `MyGroup::MyEngine` .

```
swnode unsetparam MyGroup::MyEngine buildType
```

Generic engine properties

The generic engine properties are grouped into the following sections:

General section

The following table describes the engine properties in the General section:

ECC / swnode	Deployment spec
Name / name	The name displayed in the Engine Control Center. You can only change the value through the corresponding deploy specification.
Component Description / description	Description information displayed in the Engine Control Center

Command Line section

The following table describes the engine properties in the Command Line section:

ECC / swnode	Description
Executable Type / buildType	The executable type to execute. This is either DEVELOPMENT, PRODUCTION, or ALL.
Development Executable / developmentExecutable	The path to the development executable file for the engine. This is automatically configured by the ECC when you add an engine. You should not edit this property.
Production Executable / productionExecutable	The path to the production executable file for the engine. This is automatically configured by the ECC when you add an engine. You should not edit this property.
Executable Arguments / engineArguments	Arguments passed to the executable on its command line when the ECC starts the process. This is automatically configured by the ECC when you add an engine. You should not edit this property.

Process Control section

The following table describes the engine properties in the Process Control section:

ECC / swnode	Description
Automatic Startup / autoStart	True if this engine should be started when the coordinator starts up.
Automatic Restart / restart	True if this engine should restart after it is stopped to perform node recovery.
Automatic Reload / reload	True if this engine should be reloaded if modified during a deployment specification load.
Max Fail Limit / maxFailCount	Number of consecutive times an engine can fail before the System Coordinator will no longer restart it.
Thread Termination Timer / killTimeout	Time to wait for voluntary thread shutdown before thread kill is issued.
Engine Termination Timer / abortTimeout	Time to wait for graceful engine shutdown before forced termination.
Global Engine Termination Timer / engineKillTimeout	This is a global limit on the time to wait for any engine to voluntary shutdown. KIS engines have additional properties to control engine termination.

ECC / swnode	Description
	See killTimout and abortTimeout. This value should be larger than the abortTimeout set for any engine.
System Exception Handling / exitOnSystemException	Indicates whether to exit an engine if an unhandled system exception is detected. A value of no indicates that the engine should not be shutdown, a value of yes indicates that the component should be shutdown and node recovery performed. If this property is configured to have a value of no, the runtime will catch the system exception and discard the event that caused it to occur.
Low Memory Warning Level / lowMemoryWarningLevel	A warning will be written to the engine log file when the shared memory detects that this percentage of shared memory has been used. The message is only printed once per invocation of an engine.
Flusher Timer Interval / flushRate	The flusher time interval in seconds. The flusher wakes up every flusher time interval to look for objects to flush. A value of 0 disables the flusher.
Number of Types Per Flush / numTypesPerFlush	This is the number of types the flusher will look at for flush candidates every time it wakes up. If this number if less than the total number of types installed on the node, the flusher will sequentially walk through all of the types before starting over.
Pause Process in Startup / debugPause	Whether to pause the engine in startup after the dlopen of all the libraries, but before any of the initialization code has been run. Useful for attaching a debugger before the engine starts running.

Environment section

The following table describes the engine properties in the Environment section:

ECC / swnode	Description
Runtime Directory / workingDirectory	Engine working directory. Path must be absolute.
Inherit Environment / inheritEnvironment	Does the engine inherit the coordinator's environment?
Environment / environment\<variablename>	Environment variables for engine. If you specify an environment variable in this property, it replaces that variable in the parent environment.

Thread Pool section

The following table describes the engine properties in the Thread Pool section:

ECC / swnode	Description
Initial Pool Size / numReadChannels	Initial thread pool size on engine startup. The runtime starts this many threads when the engine is started.
Min Pool Size / minReadChannels	Minimum thread pool size. The number of threads running in this engine never decreases below this number.
Max Pool Size / maxReadChannels	Maximum thread pool size. The total number of threads in the engine's thread pool never increases beyond this number.
Pool Increment / readChannelInc	Thread pool growth increment. When the thread pool is grown, it always grows by this number.
Service Period / servicePeriod	Interval time, in seconds, to poll thread pool for starvation. If thread starvation is detected, the thread pool is grown by the pool increment amount.

Transaction section

The following table describes the engine properties in the Transaction section:

ECC / swnode	Description
Deadlock Backoff / deadlockBackoffMSec	Time, in milliseconds, to delay transaction replay in the event of a deadlock.

Tracing section

The following table describes the engine properties in the Tracing section:

ECC / swnode	Description
Trace Enable / traceEnable	Enable or disable engine tracing. Excessive tracing is detrimental to engine performance.
Trace File / traceFile	Path to engine trace file.
Truncate on Startup / traceTruncate	Whether to truncate the trace file when the engine starts up.
Max Trace File Size / traceSize	Maximum trace file size before truncation. A value of 0 indicates that the trace file should not be truncated.
Trace filters / traceDebugFilter	The tracing section ends with a long list of individual tracing categories and settings for each: debug, info, warning, and fatal. These settings let you enable or disable trace message generation by category and severity.

Trace files

KIS engines and the System Coordinator can generate trace messages into log files in the node runtime directory. These messages can be useful for debugging applications or diagnosing system performance. Each engine uses its own trace file, as does the System Coordinator. You can use the engine management tools to set the filename, maximum size, and other characteristics of trace files.

An engine (or the system coordinator) only generates the types of trace messages that it is set to generate. You can use the engine management tools to control the trace level and trace filters used when generating a trace.

The Engine Control Center provides support for monitoring log files on a KIS node (see the section called “ Engine Control Center ” on page 24).

For more on configuring trace files and filters, see the section called “ Tracing section ” on page 40.

Trace severity

There are four trace severities defined in KIS. In decreasing order of severity, they are: fatal, warning, info, and debug. You can set the overall level of tracing for the engine or coordinator.

Generating trace files can have a large negative effect on system performance. Enabling debug tracing will cause serious system performance degradation.

Trace filters

In addition to the overall level of trace messages, you can also configure which parts of KIS can generate each level of trace message.

For each individual engine, and for the System Coordinator, you can set:

- the trace level
- the trace severity

Engine lifecycle events

There are several IDLos properties that you can use to trigger operations on engine lifecycle events.

[packageInitialize]

The system invokes `[packageinitialize]` operations when an engine starts, but before events are dispatched. Thus, such operations are guaranteed to execute before the engine starts processing other work. However, `[packageInitialize]` operations are restricted to two-way operations without parameters and return value.

[initialize]

The system invokes `[initialize]` operations when an engine starts, but after the event bus starts. Thus, their invocation may occur after the engine already begins to process work. This limits their usefulness for initialization. However, `[initialize]` operations do not suffer the same restrictions as `[packageInitialize]` operations.

[unload]

The system invokes `[unload]` operations when a dynamic engine unloads its dynamic components. You can use `[unload]` operations to free any resources.

[reload]

The system invokes `[reload]` operations when a dynamic engine loads its dynamic components. You can use `[reload]` operations to reinitialize thread resources.

[terminate]

The system invokes `[terminate]` operations during ordinary engine shutdown, not when an engine faults, or when an engine is killed in order to perform recovery.

The sequence of operations for various lifecycle events are as follows:

Lifecycle event	Operations invoked
Engine Started	[packageInitialize]
	[initialize]
Dynamic Engine Re-loaded	[unload]
	[reload]
Engine Stopped	[terminate]

4

KIS Monitor

The Kabira Infrastructure Server (KIS) Monitor is a powerful diagnostic tool that allows you to interactively decode and view any data structure in shared memory. For example it is possible to display user objects in shared memory making it extremely helpful in monitoring and debugging applications that you develop with KIS.

This chapter contains the following sections:

- the section called “ Starting the KIS Monitor ” on page 43
- the section called “ Working with the KIS Monitor ” on page 44
- the section called “ Viewing types and instances ” on page 47
- the section called “ An example model view ” on page 50

Starting the KIS Monitor

You start the KIS Monitor by entering the following command at a shell prompt:

```
swmon [shared memory filename]
```

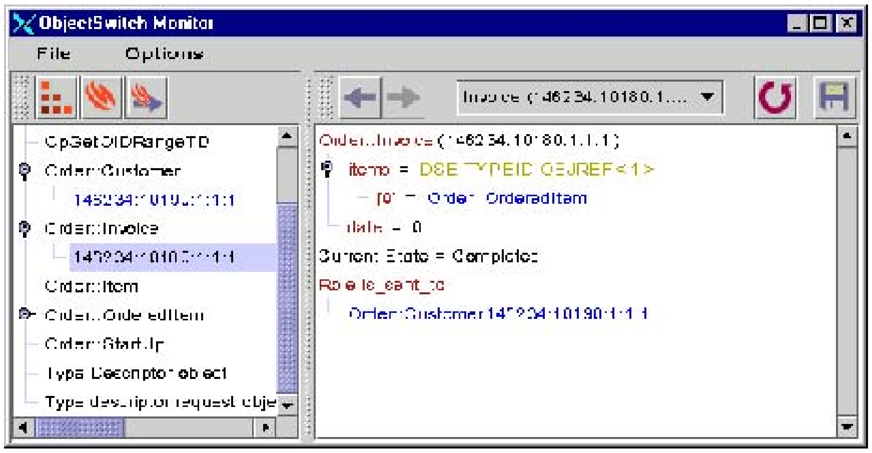
Your path should be set correctly before you use KIS commands. See “User Environment” on page 3

Specifying a shared memory file in the command line is optional; you can attach a shared memory file after launching the KIS Monitor.

Working with the KIS Monitor

The KIS Monitor display is divided into two panes. The left pane is a “table of contents,” or index, of all the objects in the system that apply to the current view. The right pane displays the contents of the currently selected object.

The entities and extents that are listed in the left pane function as object containers. This is indicated by the toggle icon preceding the respective identifier. By clicking on the corresponding toggle icon, you can expand and collapse a container; this displays and hides the object references of all objects of that type in shared memory.



```
CpGetOIDRangeTD
Order::Customer
    146234::0190::1:1
Order::Invoice
    146234::0180::1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
Order::Invoice(146234.10180.1.1.1)
items = OSE TYPEID OBJREF<1>
    [0] = Order::OrderedItem
date = 0
Current State = Completed
Role is_sent_to
    Order::Customer 146234:10190:1:1:1
```

Figure 4.1. KIS Monitor

Color coding

The following table describes the color coding used to represent the different types of elements:

Color	Type
blue	offset or object reference
gold	type reference
red	slot in an object’s shared memory image
green	string
black	literal value or primitive type

File menu

Provides commands for managing files associated with the KIS Monitor as well as for closing the KIS Monitor.

Attach Memory... lets you to attach a shared memory file; a file selection dialog opens allowing you to select a shared memory file, typically named “ossm”.

Properties... displays the fully qualified name of the shared memory file currently attached to the KIS Monitor.

Open Log File... selects a file for logging the contents of the right pane of the KIS Monitor. (See the section called “ Writing to a log file ” on page 46.)

Close Log File unselects the log file, if one is selected.

Close exits the KIS Monitor.

Options menu

The commands in the Options menu let you show/hide detail information and set the level of decoding as well as specify the order in which the types are displayed.

Filter Generated Types filters out objects that do not correspond to entities from the user model.

Show Type ID displays types together with their type ID. (See "Type ID and object reference" in the following section for a detailed description of type IDs.)

Show Cardinality displays types together with the number of corresponding objects found in shared memory, which is very helpful in determining the number of instances of a type in shared memory.

Decoding Level... enables the interpretation of different data structures. There are three levels of decoding: Model, System, and Primitive.

- Model decoding—shows object instances as close to the modeled objects as possible; this decoding level will generally be the most valuable in viewing user objects.
- System decoding—displays object instances in terms of their system runtime data structures; however, still automatically decodes certain complex types like strings and arrays.
- Primitive decoding—only interprets primitive types; provides no decoding of complex types.

Sort... lets you pick how you want to sort the types in the left pane

- By Name—sorts alphabetically.
- By Type ID—sorts according to type IDs.

Perferences allows you to set font preferences for the monitor that are saved so that they are in affect the next time the monitor is run.

Buttons—Views

The KIS Monitor offers three different views of the data structures found in shared memory: the Allocator, Model and Transaction views. Those views correspond to the three buttons above the left pane of the KIS Monitor:

The Allocator View displays all allocated system objects. A system object is used to support the operation of the KIS. It is not a modeled object. The System View is geared towards the KIS field or support engineer. As an application developer, you will probably not be interested in the data structures presented in this view.

The Model View displays user objects in shared memory, which are generated by KIS from the user model, for example, entities, extents, typedefs, and structs. This view will be of great assistance in monitoring your applications.

The Transaction View is only marginally of interest for a running application; normally, by the time you click on a transaction, it's already complete. This view is intended for developers diagnosing hung systems.

Navigating shared memory

Offsets and object references as well as type references are all represented in the right pane as hotlinks; clicking on one displays the object in the right pane.

You can quickly navigate to related objects by just clicking on the object references of the respective roles and attributes; or traverse type descriptor definitions by clicking on the various type references.

History buttons and list

The KIS Monitor records a history of the types and instances that it previously displayed. You can navigate the history using the Prior and Next buttons and the drop-down history list, all of which appear above the right pane of the KIS Monitor.

You can use the

Prior and Next buttons to navigate through the history of previously displayed types and instances.

Alternatively, you can use t

he drop-down history to select a particular type or instance that was previously displayed. Clicking on the control drops down a scrollable history, and clicking on one of the items in the history opens the item's decoded representation.

Reloading the shared memory file

The types and objects displayed by the KIS Monitor represent a snapshot of shared memory at the instance of navigating to the currently displayed type or object. You update the display with the current state of shared memory by clicking on the Reload button, by clicking on a view button, or clicking on another type or object reference.

Writing to a log file

You can write the contents of the right pane to a log file with the Save button, appending the current contents to the log file. If a log file is not currently selected, clicking on the Save button opens a file selection dialog, allowing you to select a log file before saving. (See the section called “ Open Log File... ” on page 45.)

Viewing types and instances

The left pane of the KIS Monitor lists the names of all the types that apply to the current view. When shared memory contains objects of a particular type, the Design Center lists references to those objects (object IDs) under the name of the type and displays a toggle icon in front of the type’s name. Clicking on the toggle icon shows and hides the object references of that type.

Figure 4.2 illustrates the Model View where the `Order::Customer` and `Order::Invoice` types have been expanded to show references to objects of those types.

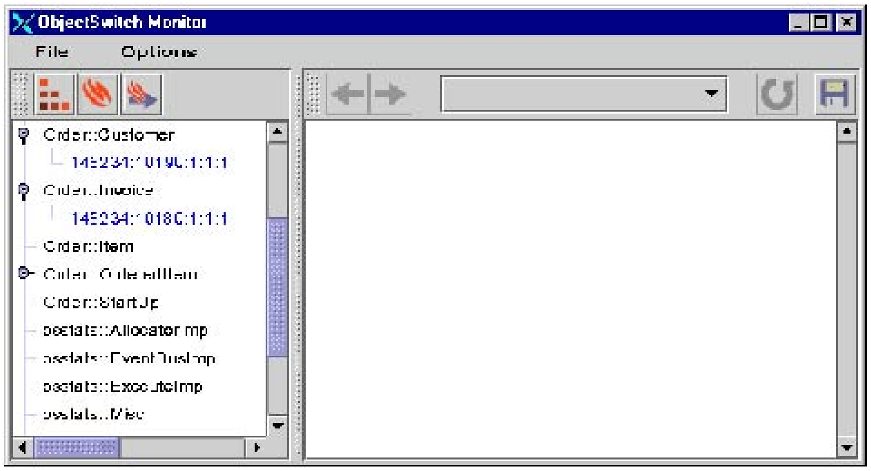


Figure 4.2. Types and object references in the left pane

Type ID and object reference

A type ID is used to identify each described type. These type IDs are based on the fully qualified type name.

Object references are based on type IDs, and uniquely identify an instance throughout all KIS nodes. An object reference is made up of four parts which are described below.

type ID	implementation	location	OID

The type ID is a two-part construction, usually given in the form `dddd:tttt` so that the KIS Monitor displays these fields in the form:

```
dddd:tttt:iiii:llll:oooo
```

Type ID This is generated by the object compiler. The type ID is unique within a repository and identifies a KIS type.

Implementation This is the object implementation—either transient or persistent, using some implementation adapter. It will be in the range 1-4, where:

1	The default KIS transient implementation.
2	Sybase persistent implementation.
3	Informix persistent implementation.
4	Oracle persistent implementation.

Location This is the location code of the KIS node where an instance is created.

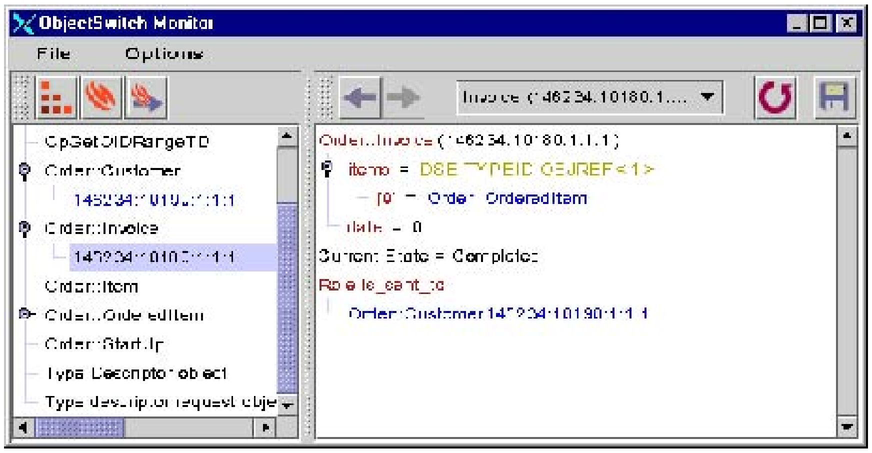
OID This is generated at runtime, and only has to be unique within the scope of all the above qualifiers. For persistent objects that are uniquely keyed with a numeric value, it is generated by the backing database. For all other objects, KIS generates the OID.

Decoded Objects

When you click on an object reference, the KIS Monitor displays the decoded representation of that object in the right pane, which can include:

- Attributes with their current values
- Roles with references to the related objects
- The current state

The attributes can contain nested values, a sequence, struct, or array, which you can expand and view member or elements by clicking on the toggle icon.



```
Order::Invoice (146234.10180.1.1.1)
  items = DSE_TYPEID OBJREF<1>
    [0] = Order::OrderedItem
  date = 0
Current State = Completed
Role is_sent_to
  Order::Customer 146234:10180:1:1:1
```

Figure 4.3. Decoded objects

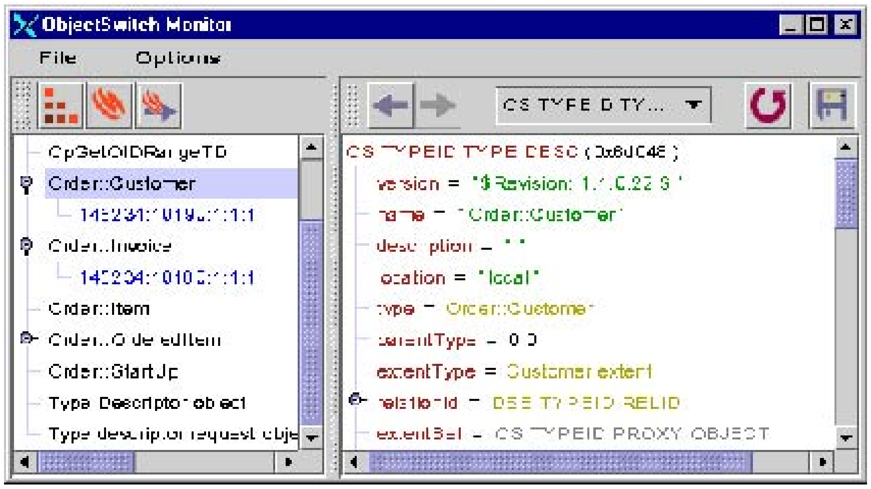
Current state If a state machine is defined for an object, the KIS Monitor displays the object's current state. Now by definition of a state machine (see the “State machines” section of the “Developer's Guide”), an action is executed upon entering each state. However, the current state is not updated until the corresponding transaction has been committed.

Thus, if you halt execution during an object's state action and view the decoded representation of that object in the Transaction view, the displayed current state is not the state being entered but the last committed state.

Decoded Types

KIS maintains type information in shared memory using a type descriptor object to describe each type. Type descriptors are meta-data, and as such can describe the structure of any object in shared memory. Each such object describes in a generic fashion how the objects of the respective type are constructed. Every type in the system has a type descriptor.

When you select a type name in the left pane (or right pane) of the KIS Monitor, the type descriptor object for that type is displayed in the right pane:



```
CpGetOIDRangeTD
Order::Customer
   142231::0190::1:1
Order::Invoice
   142234::0100::1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
CS TYPEID TYPE DESC (0x8dC4E)
  version = "$Revision: 1.1.0.22 $"
  name = "Order::Customer"
  description = ""
  location = "local"
  type = Order::Customer
  parentType = 0 0
  extentType = Customer extent
  relationId = DSE TYPEID RELID
  extentSel = CS TYPEID PROXY OBJECT
```

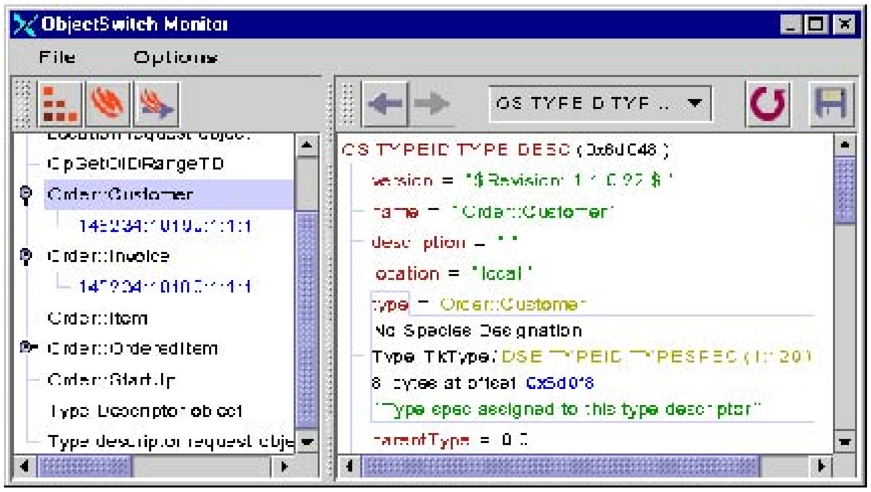
Figure 4.4. Type information

Each type descriptor objects has, for example, slots for version, name, description, type, parent type (inheritance), extent, size and species (struct, exception, dynamic, sequence, array, or interface). Only the values assigned to the different slots vary for each type. The entry `objectSize` can be used to determine the size of that object in shared memory.

Some slots reference a nested type, a struct, sequence, or array. The KIS Monitor indicates that the nested type contains values that can be viewed by preceding the slot identifier with a toggle icon. Clicking on the toggle icon expands the slot to display underneath the identifier the contents of the referenced type. The nested slots may contain still other expandable type references.

If the nested type is a sequence, the KIS Monitor displays the type reference with following angle brackets (“<>”). The KIS Monitor shows the number of elements contained in the sequence inside the angle brackets.

Clicking on any slot identifier in the right pane shows/hides the technical details of that slot. Figure 4.5 shows the expanded `type` slot identifier:



```
Location request object
CpGetOIDRangeTD
Order::Customer
    145234::0190::1:1
Order::Invoice
    145234::0100::1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
OS TYPEID TYPE DESC (0x8dC48)
    version = "$Revision: 1.10.27 $"
    name = "Order::Customer"
    description = ""
    location = "local"
    type = Order::Customer
    No Species Designation
    Type TkType: OSE TYPEID TYPESPEC (1::20)
    8 bytes at offset 0x5d0*8
    "Type spec assigned to this type descriptor"
    parentType = 0.0
```

Figure 4.5. Slot details

Slot details include:

- species (an enumeration of the “kind” of object)
- type ID (with a hotlink to the type descriptor)
- slot size
- physical memory address
- slot description

An example model view

This section introduces an example model and illustrates how the KIS Monitor represents the types, objects, and transactions.

The example model

The model consists of two main entities, `Invoice` and `Customer`, which are related through relationship `R1`.

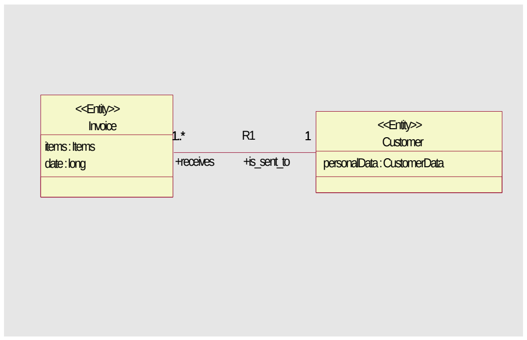


Figure 4.6. The general user model

Invoice The items included in an `Invoice` object, represented by the `items` attribute, are contained in a sequence of `OrderedItem` .

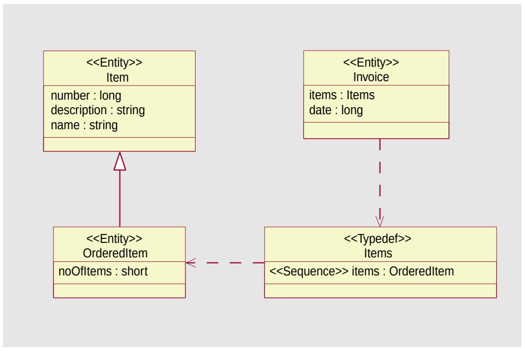


Figure 4.7. Invoice

`OrderedItem` is a specialized type of the `Item` entity, additionally giving the quantity for each specific item being ordered. The date of the `Invoice` object is given by the `date` attribute.

As shown in Figure 4.8, an `Invoice` object has 5 different states: `OrderTaken` , `Billing` , `Completed` , `Cancelled` , and `PoliceNotified` . There is a transition from the initial state `OrderTaken` to `Billing` when an invoice receives the `billCustomer` signal, which requires two arguments, one of type `Items` , a sequence or `OrderedItem` , and the other an object of type `Customer` . The action executed when entering the `Billing` state performs a credit check on the customer and depending on the outcome, sends a signal `goodCustomer` , `badCustomer` , or `uglyCustomer` , which causes a transition to either the `Completed` , `Cancelled` , or `PoliceNotified` state, respectively.

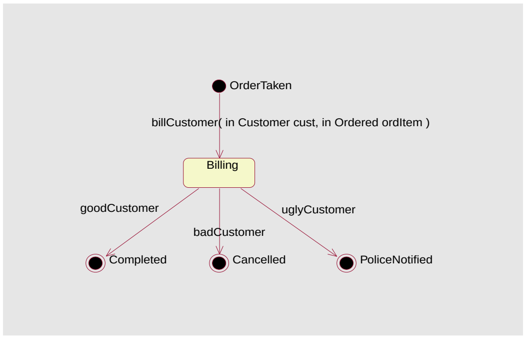


Figure 4.8. The Invoice state model

Customer The `Customer` entity contains an attribute defined as a struct `CustomerData` , which specifies the personal data for one customer, including a member defined with the help of an enumeration, `CustomerType` , specifying the type of customer, `good` , `bad` , or `ugly` .

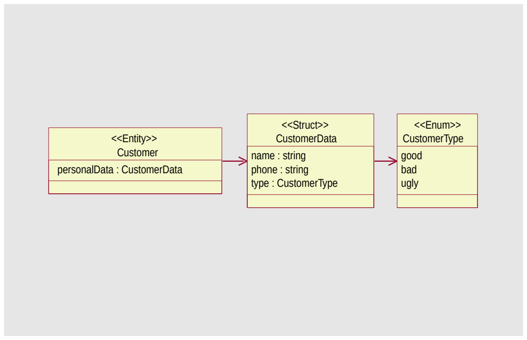


Figure 4.9. Customer

You can create an `OrderedItem` object and add it to a sequence of `OrderedItem` objects with action language like the following:

```
declare Items ordItems;
declare OrderedItem ordItem;
create ordItem;
ordItem.number = 01234;
ordItem.description = "Six-string electric guitar...";
ordItem.name = "Fender Stratocaster";
ordItem.noOfItems = 1;
ordItems[ordItems.length] = ordItem;
```

The following action language code creates a new customer:

```
 // John's data
 declare CustomerData johnData;
 johnData.name = "John Lennon";
 johnData.phone = "415/567-3425";
 declare Customer john;
 create john;
 john.personalData = johnData;
```

An `Invoice` object is created with the next action language statements:

```
declare Invoice inv;
create inv;
```

When you create an `Invoice` object, it goes into the `OrderTaken` state and awaits the `billCustomer` signal, to bill a customer for a number of ordered items. For example, the customer `john` is billed for some ordered items `ordItems` with the following code:

```
inv.billCustomer(john, ordItems);
```

The `Invoice` object goes into the `Billing` state and executes the following action language, acting on the `cust` argument, of type `Customer` , and the `ordItems` argument, of type `Items` :

```
declare ::Order::CustomerData custData;
declare ::Order::Customer localCust;
localCust = cust;
custData = localCust.personalData;
declare ::Credit::CreditRating crdrating;
create crdrating;
```

```
// Stops execution for 20 seconds
// allowing the current transaction to be viewed.
SWBuiltIn::nanoSleep(20,0);
crdrating.checkRating(custData.name);
localCust.rating = crdrating;

if (crdrating.rating == 0)
{
    custData.type = good;
    self.items = ordItems;
    relate localCust receives self;
    relate self is_sent_to localCust;
    self.goodCustomer();
}
else if (crdrating.rating ==1)
{
    custData.type = bad;
    self.badCustomer();
}
else
{
    custData.type = ugly;
    self.uglyCustomer();
}
```

A credit check is performed on the customer using the customer's name and the customer type is updated accordingly.

- If the credit check is good (= 0), the ordered items are assigned to the invoice, the relationship is set up between the invoice and the customer, and with the `goodCustomer` signal, the invoice goes into the `Completed` state.
- If the credit check is bad (= 1), the invoice goes into the `Cancelled` state with the `badCustomer` signal.
- Otherwise, the invoice goes into the `PoliceNotified` state with the `uglyCustomer` signal.

The action language additionally contains a built-in function, `nanoSleep`, that stops execution for 20 seconds to provide a glimpse of the Transaction View.

Viewing the model

Attaching the shared memory file to the KIS Monitor immediately after starting the model application and clicking on the Transaction View produces a KIS Monitor display similar to the one appearing in Figure 4.10. The presence of an engine instance indicates that execution of this engine has been stopped, in this case due to the built-in function `nanoSleep`.

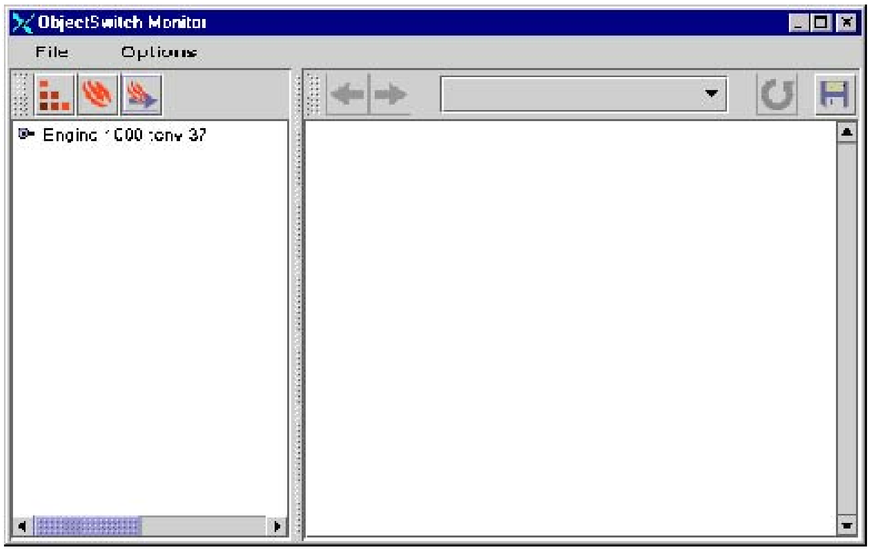
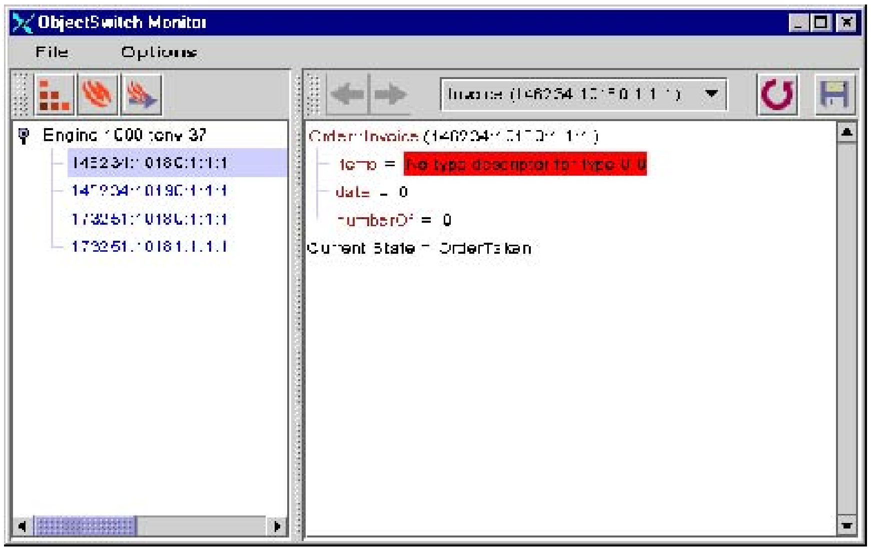


Figure 4.10. Transaction View

Clicking on the toggle icon in front of the engine instance expands the transaction container. The display in the left pane lists an object reference for each object created within the current transaction as well as their extent. Clicking on the object reference for the `Invoice` object displays the decoded representation of that object, as shown in Figure 4.11.



```
Order::Invoice ([illegible])
  temp = No type descriptor for type [illegible]
  date = 0
  numberOf = 0
Current State = OrderTaken
```

Figure 4.11. Invoice object in stopped transaction

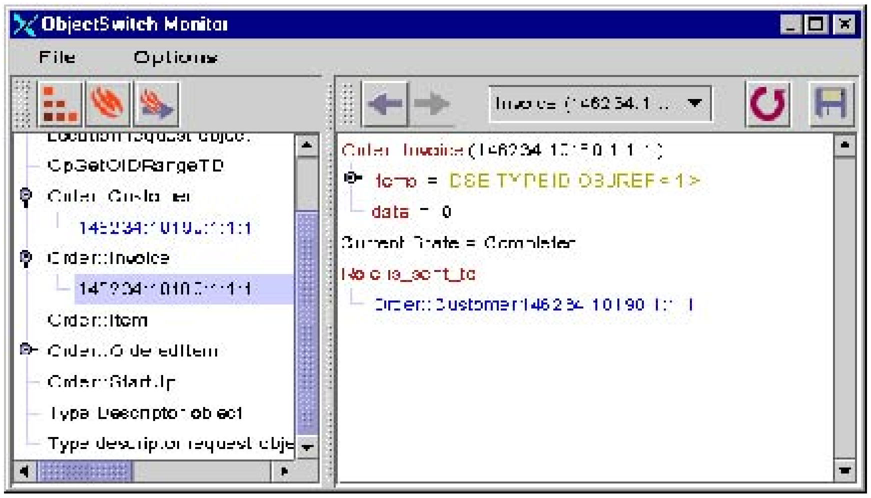
Since the `nanoSleep` function stops execution before any attribute value assignments, you see the default attribute values. The current state is also displayed.

After 20 seconds execution continues and because customer `john` has a good credit rating,

- the sequence of `OrderedItem`, `ordItems`, is assigned to the `items` attribute of the `Invoice` object

- the `Invoice` object is related to the `john` object via the `R1` relationship, and vice versa
- the type attribute of the customer `john` is set to `good`
- there is a transition of the `Invoice` object's state from `Billing` to `Completed`

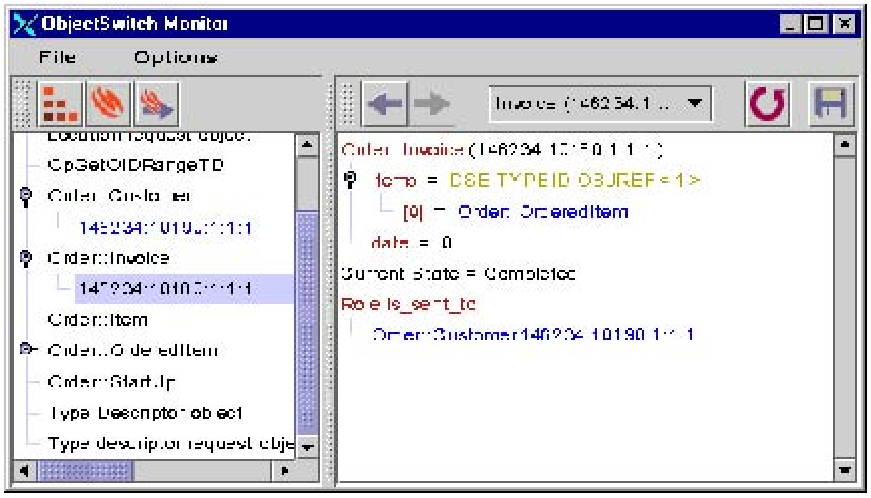
You can view this from the Model View after clicking on the corresponding button. Clicking on the toggle icon in front of the `Invoice` type expands its contents and clicking on the `Invoice` object reference under the Invoice type shows the decoded representation of the object, as in Figure 4.12.



```
Order::Invoice(146234:10190:1:1)
  items = CSE TYPEID OBJREF<1>
  date = 0
Current State = Completed
Role is_sent_to
  Order::Customer146234:10190:1:1
```

Figure 4.12. Invoice object in the Completed state

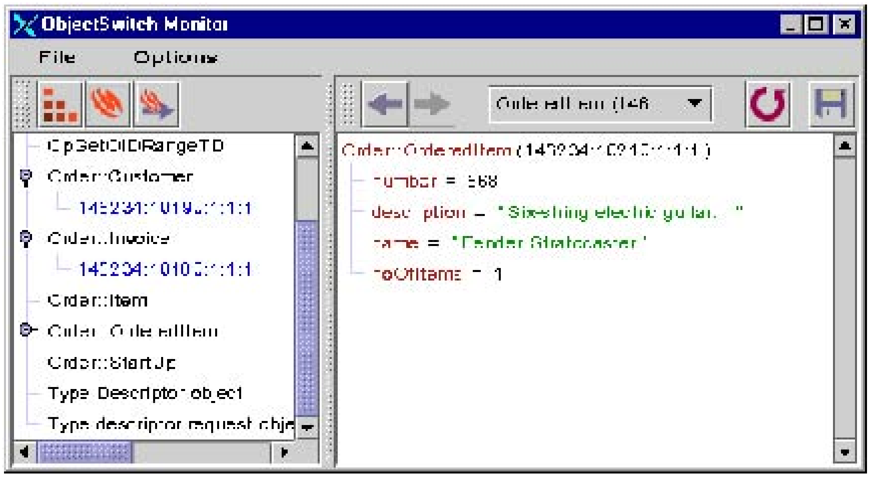
Since the `items` attribute is a nested type, a sequence of `OrderedItem`, there is a toggle icon in front of it. Clicking on the toggle icon expands the sequence so that you can view the individual elements, as shown in Figure 4.13.



```
Order::Invoice(146234:10190:1:1)
  items = CSE TYPEID OBJREF<1>
    [0] = Order::OrderedItem
  date = 0
Current State = Completed
Role is_sent_to
  Order::Customer146234:10190:1:1
```

Figure 4.13. Invoice object with expanded items

The sequence contains only one element, which is represented by the hotlinked object reference. Clicking on the object reference opens the decoded representation of the `OrderedItem` object.



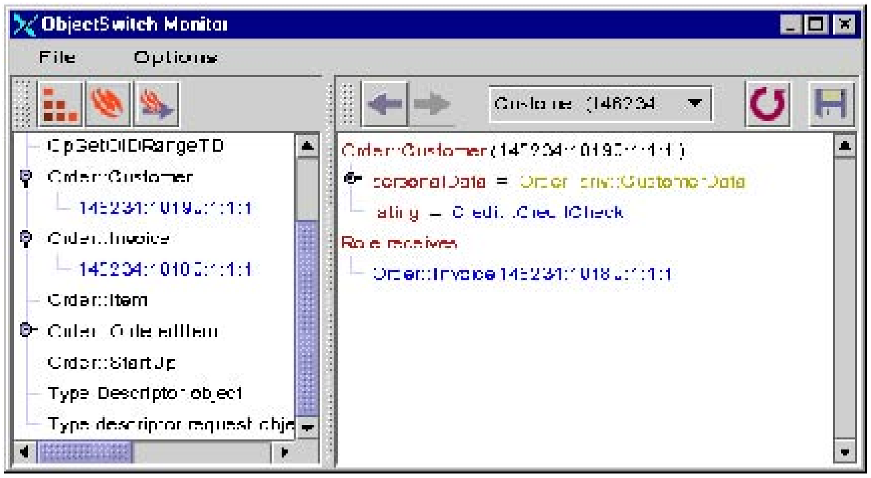
```
CpGetOIDRangeTD
Order::Customer
  145234::019...::1:1
Order::Invoice
  145234::010...::1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
Order::OrderedItem (145234::021...::1:1 )
  number = 568
  description = "Six-string electric guitar. "
  name = "Fender Stratocaster"
  noOfItems = 1
```

Figure 4.14. OrderedItem object

Clicking on the Previous button returns to the representation of the `Invoice` object displayed in Figure 4.13. The object reference under the `is_sent_to` is a hotlink to the `Customer` object `john` created above.

Clicking on the object reference under the `is_sent_to` role opens the decoded representation of that object in the right pane, as shown in Figure 4.15.

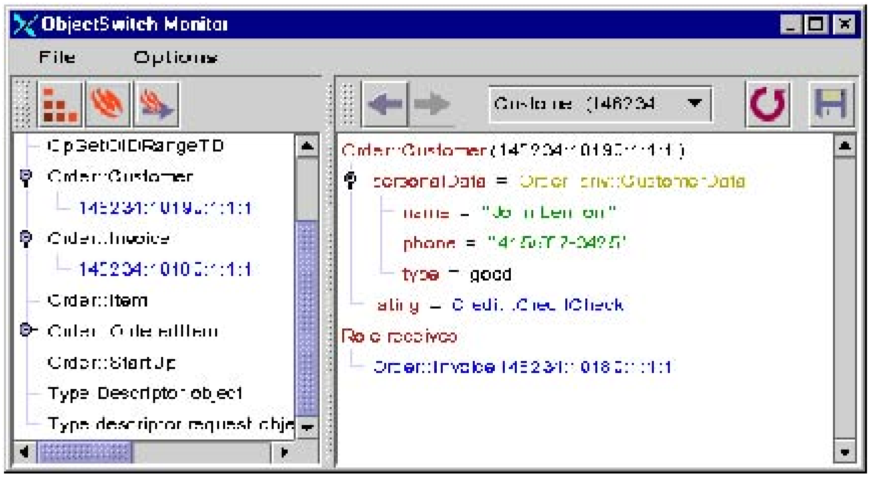


```
CpGetOIDRangeTD
Order::Customer
  145234::019...::1:1
Order::Invoice
  145234::010...::1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
Order::Customer (145234::019...::1:1 )
  personalData = Order ens::CustomerData
  rating = Credit::CreditCheck
Role receives
  Order::Invoice 145234::018...::1:1
```

Figure 4.15. Customer object

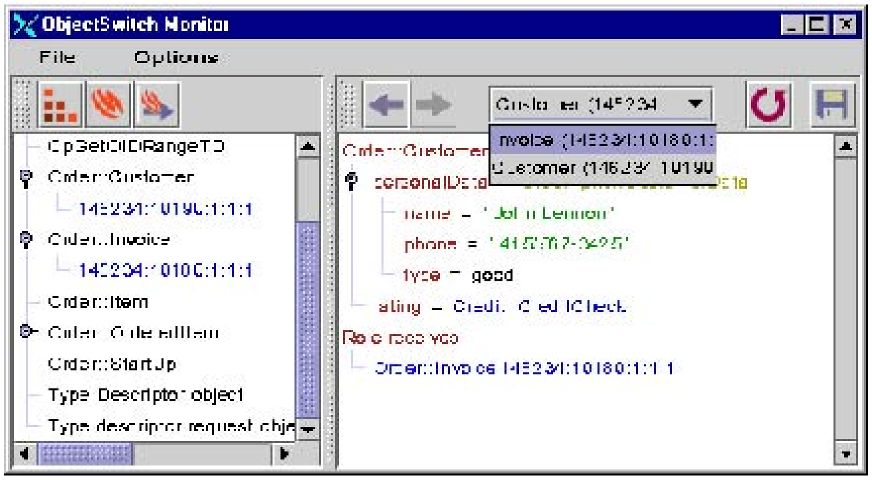
Clicking on the toggle icon preceding the `personalData` slot identifier expands the slot to display the contents of the nested type.



```
Order::Customer (145234::0190:1:1:1)
personalData = Order::Customer::CustomerData
    name = "John Lennon"
    phone = "415/767-0425"
    type = good
  rating = CreditCheck::CreditCheck
Role receives
  Order::Invoice 145234::0180:1:1:1
```

Figure 4.16. A customer object with expanded struct

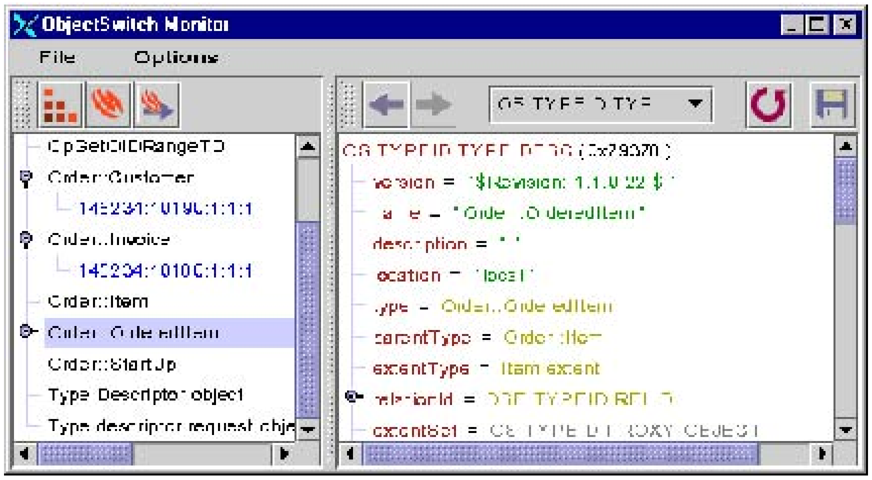
The object reference appearing underneath the `receives` role in Figure 4.16 is a hotlink back to decoded representation the `Invoice` object. Clicking on the Prior button returns to decoded representation of the `Invoice` object in the right pane, too. Likewise, you can use the drop-down history, as shown in Figure 4.17, to return to the representation of the `Invoice` object displayed in Figure 4.13.



```
Order::Customer
personalData
    name = "John Lennon"
    phone = "415/767-0425"
    type = good
  rating = CreditCheck::CreditCheck
Role receives
  Order::Invoice 145234:10180:1:1:1
```

Figure 4.17. An invoice object

Finally, clicking on the `Order::OrderedItem` type reference in the left pane opens the decoded type, as shown in Figure 4.18. Notice the contents of the `parentType` slot, which indicates that `OrderedItem` is a descendent of `Item`.



```
OpGetOIDRangeTD
Order::Customer
    145234::0190:1:1:1
Order::Invoice
    145234::0100:1:1:1
Order::Item
Order::OrderedItem
Order::StartUp
Type Descriptor object
Type descriptor request obje
```

```
OS TYPEID TYPE DESC (0x79070 )
  version = "$Revision: 1.1.0.22 $ "
  name = "Order::OrderedItem"
  description = " "
  location = "local"
  type = Order::OrderedItem
  parentType = Order::Item
  extentType = Item extent
  relationId = OSF TYPEID REL D
  extentSet = OS TYPE DT ROXY OBJECT
```

Figure 4.18. The OrderedItem type

Index